XIAMEN UNIVERSITY MALAYSIA

Course Code	:	CST207
Course Name	:	Design and Analysis of Algorithms
Lecturer	:	Dr. Mohammed N. M. Ali
Academic Session	:	2025/09
Assessment Title	:	AI-Driven Sorting Algorithm Optimizer
Submission Due Date	:	28th December 2025

Prepared by :

Student ID	Student Name
DSC2409006	Cui Zeyu
DSC2409010	Fan Li
DSC2409012	He Bingguang
DSC2409042	Yu Fangrui
DSC2409043	Yu Mingquan

Date Received :

Feedback from Lecturer:

Mark:

Own Work Declaration

I/We hereby understand my/our work would be checked for plagiarism or other misconduct, and the softcopy would be saved for future comparison(s).

I/We hereby confirm that all the references or sources of citations have been correctly listed or presented and I/we clearly understand the serious consequence caused by any intentional or unintentional misconduct.

This work is not made on any work of other students (past or present), and it has not been submitted to any other courses or institutions before.

Signature:

崔泽禹 范州 何秉光 谢芳瑞 余铭泉

Date: 21st December 2025

AI-Driven Sorting Algorithm Optimizer Report

CST207 Design and Analysis of Algorithms Group Project

Academic Session: 2025/09

Oral Presentation Video Link

`https://drive.google.com/drive/folders/1YNGKs7LQVs_FJW2VaJKDxGBAhHr5GaCm`

Table of Contents

Group Contribution Table

Student ID	Name	Specific Contribution Description
DSC2409006	Cui Zeyu	Implement Result Visualization GUI with Qt, Write Report and Edit Oral Presentation Video
DSC2409010	Fan Li	Implement AI Model with kNN and Decision Tree
DSC2409012	He Bingguang	Implement AI Model with kNN and Decision Tree
DSC2409042	Yu Fangrui	Implement Dataset Generation for All Types and Testing
DSC2409043	Yu Mingquan	Implement Sorting Algorithms and Measure Performance

1 Introduction

In modern software engineering, sorting is a fundamental operation whose performance critically depends on the characteristics of the dataset—such as size, order (sortedness), and element uniqueness. While standard libraries often use a one-size-fits-all approach (typically Quicksort or Introsort), no single algorithm is optimal for every scenario.

The goal of this project is to design and implement an **AI-Driven Sorting Algorithm Optimizer**. This system acts as an intelligent library that analyzes the input dataset's features and automatically selects the most efficient sorting algorithm. By integrating Artificial Intelligence (Decision Tree model) with traditional algorithmic analysis, we demonstrate how heuristics can optimize computational resource usage.

2 System Overview

In Figuire 1, we illustrate the overall architecture of the system:

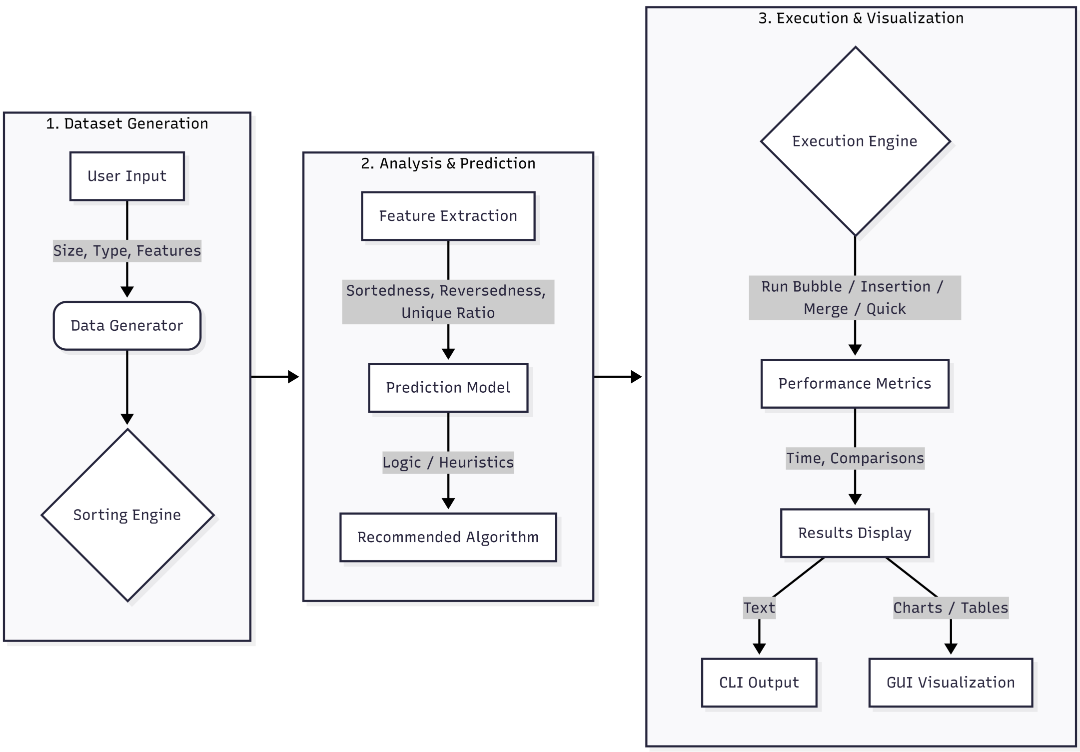


Figure 1: System Architecture Overview

The system consists of three main components:

1. **Dataset Generation:** Generate five dataset types (random, nearly sorted, reversed, few-unique, large random) with configurable size and distribution for comprehensive benchmarking. The generated datasets will be sorted by algorithms.
2. **Analysis and AI Module Prediction:** Extract features (size, sortedness, reversedness, unique ratio) and use decision-tree models to recommend the most suitable sorting algorithm. The decision tree in the AI module is constructed based on empirical observations and theoretical analysis of algorithm performance.
3. **Execution and Visualization:** Execute chosen algorithms, measure runtime and comparisons, and present results via CLI text and GUI tables. AI predictions are compared against actual performance to validate accuracy and shown visually.

3 Algorithm Design

Note that: To make the layout neater, all code in the main body of the report is uncommented. The relevant explanations will be provided in the text. Additionally, well-commented code will be added in full to the appendix.

3.1 Dataset Generation

To robustly test the algorithms, we implemented a generator capable of creating five distinct types of datasets, as required by the project specifications:

1. **Random Dataset:** Elements are generated using a random number generator with values in range $[1, size \times 10]$.

```
static vector<int> generateRandomDataset(int size) {
    vector<int> arr(size);
    srand(time(nullptr));
    for (int i = 0; i < size; i++) {
        arr[i] = 1 + rand() % (size * 10);
    }
    return arr;
}
```

2. **Nearly Sorted Dataset:** A sorted array is created, and then approximately 10% of the elements are randomly swapped to simulate slightly disordered data.

```
static vector<int> generateNearlySorted(int size) {
    vector<int> arr(size);
    for (int i = 0; i < size; i++) arr[i] = i + 1;

    int swaps = size / 10;
    srand(time(nullptr));
    for (int i = 0; i < swaps; i++) {
        int idx1 = rand() % size;
        int idx2 = rand() % size;
        swap(arr[idx1], arr[idx2]);
    }
    return arr;
}
```

3. **Reversed Dataset:** Elements in the dataset are strictly arranged in descending order $(N, N-1, \ldots, 1)$.

```
static vector<int> generateReversed(int size) {
    vector<int> arr(size);
    for (int i = 0; i < size; i++) {
        arr[i] = size - i;
    }
    return arr;
}
```

4. **Few Unique Dataset:** The dataset contains many duplicate values, generated from a small pool of unique integers. First, a pool of `uniqueCount` random values is created, then the dataset is filled by randomly selecting from this pool.

```
static vector<int> generateFewUnique(int size, int uniqueCount) {
    vector<int> uniqueValues;
    srand(time(nullptr));
    for (int i = 0; i < uniqueCount; i++) {
        uniqueValues.push_back(rand() % 100 + 1);
    }

    vector<int> arr(size);
    for (int i = 0; i < size; i++) {
        arr[i] = uniqueValues[rand() % uniqueCount];
    }
    return arr;
}
```

5. **Large Random Dataset:** A stress-test dataset with size $N \geq 10,000$ to evaluate asymptotic performance. Uses the same `generateRandomDataset()` method but enforces a minimum size constraint of 10,000 elements in the main program.

3.2 Sorting Algorithms

We implemented four classical sorting algorithms:

- **Bubble Sort ($O(N^2)$):** Included for educational comparison; effective only on very small or nearly sorted data. Features early termination optimization: if no swaps occur in a pass, the array is already sorted.

```
static void bubbleSort(vector<int>& arr, long long& comparisons) {
    int n = arr.size();
    for (int i = 0; i < n - 1; i++) {
        bool swapped = false;
        for (int j = 0; j < n - i - 1; j++) {
            comparisons++;
            if (arr[j] > arr[j + 1]) {
                swap(arr[j], arr[j + 1]);
                swapped = true;
            }
        }
        if (!swapped) break;
    }
}
```

- **Insertion Sort ($O(N^2)$):** Highly efficient for small N or nearly sorted data due to low constant factors. Each element is inserted into its correct position in the sorted prefix. For nearly sorted data, the inner while loop terminates quickly, achieving near $O(N)$ performance.

```
static void insertionSort(vector<int>& arr, long long& comparisons) {
    int n = arr.size();
    for (int i = 1; i < n; i++) {
        int key = arr[i];
        int j = i - 1;
        while (j >= 0) {
            comparisons++;
            if (arr[j] <= key) break;
            arr[j + 1] = arr[j];
            j--;
        }
        arr[j + 1] = key;
    }
}
```

- **Merge Sort ($O(N \log N)$):** A stable, divide-and-conquer algorithm that guarantees performance but requires $O(N)$ auxiliary space. The algorithm recursively divides the array, then merges sorted subarrays.

```
static void mergeSort(vector<int>& arr, int l, int r, long long&
    comparisons) {
    if (l >= r) return;
    int m = l + (r - l) / 2;
    mergeSort(arr, l, m, comparisons);
    mergeSort(arr, m + 1, r, comparisons);
    merge(arr, l, m, r, comparisons);
}

static void merge(vector<int>& arr, int l, int m, int r, long long&
    comparisons) {
    int n1 = m - l + 1;
    int n2 = r - m;
    vector<int> left(n1), right(n2);

    for (int i = 0; i < n1; i++) left[i] = arr[l + i];
    for (int j = 0; j < n2; j++) right[j] = arr[m + 1 + j];

    int i = 0, j = 0, k = l;
    while (i < n1 && j < n2) {
        comparisons++;
        if (left[i] <= right[j]) {
            arr[k++] = left[i++];
        } else {
            arr[k++] = right[j++];
        }
    }
    while (i < n1) arr[k++] = left[i++];
    while (j < n2) arr[k++] = right[j++];
}
```

- **Quick Sort ($O(N \log N)$):** *Optimization:* To prevent the worst-case $O(N^2)$ time complexity on *Reversed Datasets*, we implemented a **Randomized Pivot** strategy.

The pivot is selected randomly, ensuring $O(N \log N)$ performance on average for any input distribution.

```
static void quickSort(vector<int>& arr, int low, int high, long long&
    comparisons) {
    if (low < high) {
        int pi = partition(arr, low, high, comparisons);
        quickSort(arr, low, pi - 1, comparisons);
        quickSort(arr, pi + 1, high, comparisons);
    }
}

static int partition(vector<int>& arr, int low, int high, long long&
    comparisons) {
    int randomIndex = low + rand() % (high - low + 1);
    swap(arr[randomIndex], arr[high]);

    int pivot = arr[high];
    int i = low - 1;
    for (int j = low; j < high; j++) {
        comparisons++;
        if (arr[j] < pivot) {
            i++;
            swap(arr[i], arr[j]);
        }
    }
    swap(arr[i + 1], arr[high]);
    return i + 1;
}
```

3.3 AI Module: Decision Tree Approach

We implemented a Decision Tree to predict the best algorithm. The AI module extracts four key features from the dataset: *Size*, *Sortedness* (ratio of ascending pairs), *Reversedness* (ratio of descending pairs), and *Unique Ratio*. The feature extraction is implemented by analyzing consecutive pairs and using `unordered_set` to count unique elements:

```
static DatasetFeatures analyzeDataset(const vector<int>& data) {
    DatasetFeatures features;
    features.size = data.size();
    features.isLargeDataset = (features.size > 1000);

    long long ascendingPairs = 0, descendingPairs = 0;
    for (int i = 0; i < features.size - 1; i++) {
        if (data[i] <= data[i+1]) ascendingPairs++;
        if (data[i] >= data[i+1]) descendingPairs++;
    }

    features.sortedness = (double)ascendingPairs / (features.size - 1);
    features.reversedness = (double)descendingPairs / (features.size - 1);

    unordered_set<int> uniqueElements(data.begin(), data.end());
    features.uniqueCount = uniqueElements.size();
    features.uniqueRatio = (double)uniqueElements.size() / features.size;

    return features;
}
```

The decision logic is as follows:

1. **Rule 1 (Small Dataset):** If $Size \leq 50$, predict **Insertion Sort**. The overhead of recursion in Merge/Quick Sort outweighs the simple logic of Insertion Sort for small N.
2. **Rule 2 (Large Dataset):** If $Size > 1000$:
   - If $UniqueRatio < 0.40$, predict **Merge Sort**. Merge Sort maintains $O(N \log N)$ stability with heavy duplicates, while Quick Sort may degrade due to unbalanced partitions.
   - Otherwise, predict **Quick Sort**. Quick Sort has the smallest constant factors and better cache performance for large datasets with good uniqueness.
3. **Rule 3 (Nearly Sorted):** If $Size \in (50, 1000]$ and $Sortedness \geq 0.80$, predict **Insertion Sort**. Insertion Sort degrades to $O(N + D)$ where D is inversions, achieving near-linear performance on nearly sorted data.
4. **Rule 4 (Reversed Order):** If $Size \in (50, 1000]$ and $Reversedness \geq 0.90$, predict **Merge Sort**. Merge Sort guarantees $O(N \log N)$ worst-case performance and is unaffected by initial element order.
5. **Rule 5 (Few Unique, Medium):** If $Size \in (50, 1000]$ and $UniqueRatio < 0.40$, predict **Merge Sort**. Same reasoning as Rule 2—duplicates favor Merge Sort's stability.
6. **Rule 6 (Default):** For all other cases, predict **Quick Sort**. Quick Sort is the best general-purpose algorithm with average $O(N \log N)$, smallest constant factors, and randomized pivot to avoid worst-case scenarios.

The complete decision tree is illustrated in Figure 2:

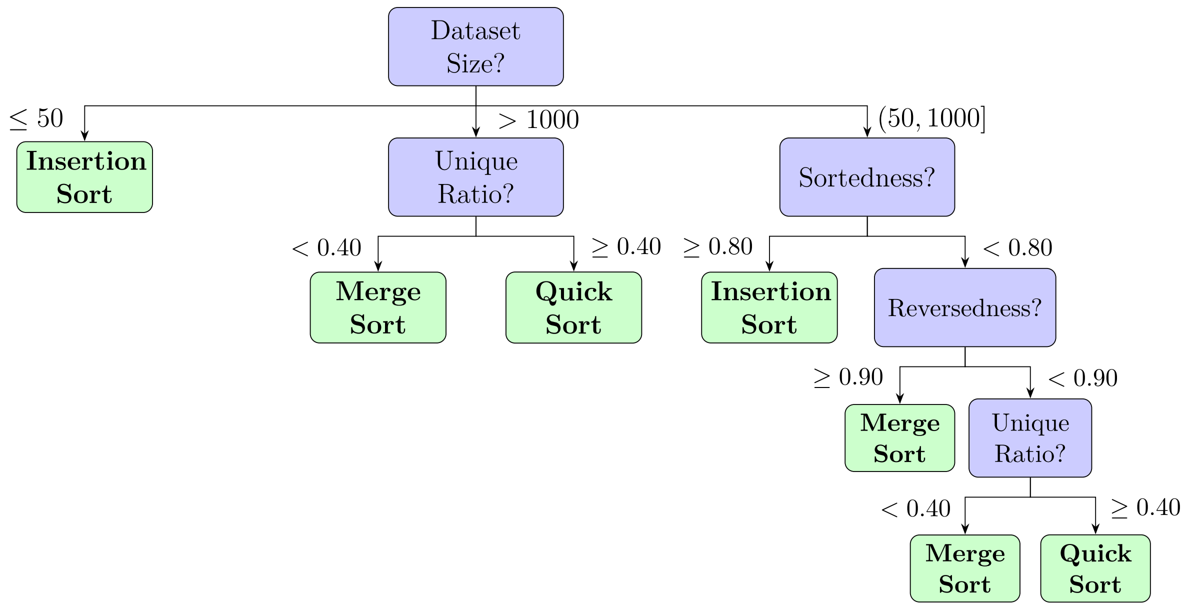


Figure 2: Decision Tree for Algorithm Selection

This decision tree effectively captures the heuristics derived from algorithmic analysis and experimental observations.

The decision tree logic for algorithm prediction is implemented as follows:

```
static AlgoType predictBestAlgorithm(const DatasetFeatures& features) {
    if (features.size <= 50) {
        return INSERTION_SORT;
    }

    if (features.isLargeDataset) {
        if (features.uniqueRatio < 0.40) {
            return MERGE_SORT;
        }
        return QUICK_SORT;
    }

    if (features.sortedness >= 0.80) {
        return INSERTION_SORT;
    }

    if (features.reversedness >= 0.90) {
        return MERGE_SORT;
    }

    if (features.uniqueRatio < 0.40) {
        return MERGE_SORT;
    }

    return QUICK_SORT;
}
```

4 Implementation Details

4.1 Performance Measurement

The system measures two metrics for every run:

- **Comparisons:** A `long long` counter passed by reference increments on every element comparison.
- **Execution Time:** Measured using `std::chrono::high_resolution_clock` in milliseconds.

The `runSort` method encapsulates both measurement processes. It records the start time before calling the sorting algorithm, passes the `comparisons` counter by reference to be incremented during execution, and then calculates the elapsed time:

```
static SortMetrics runSort(AlgoType type, vector<int> data) {
    SortMetrics metrics;
    metrics.algoName = getAlgoName(type);
    metrics.comparisons = 0;

    auto start = chrono::high_resolution_clock::now();

    switch (type) {
        case BUBBLE_SORT: bubbleSort(data, metrics.comparisons); break;
        case INSERTION_SORT: insertionSort(data, metrics.comparisons);
    break;
```

```cpp
        case MERGE_SORT: mergeSort(data, 0, data.size() - 1, metrics.comparisons); break;
        case QUICK_SORT: quickSort(data, 0, data.size() - 1, metrics.comparisons); break;
    }

    auto end = chrono::high_resolution_clock::now();
    chrono::duration<double, milli> duration = end - start;
    metrics.executionTimeMs = duration.count();

    return metrics;
}
```

4.2 Safety and Optimization

To prevent system unresponsiveness, the implementation includes a safety check: if the dataset size exceeds 1,000 elements, $O(N^2)$ algorithms (Bubble and Insertion Sort) are automatically skipped. This ensures that "Large Random Datasets" (e.g., $N = 100,000$) only benchmark efficient $O(N \log N)$ algorithms.

When an algorithm is skipped, the `comparisons` and `executionTimeMs` are set to -1 as a marker, and the result table displays "(Skipped)" instead of numerical values. In the main execution loop, before running each algorithm, the system checks whether the algorithm's complexity and dataset size combination would cause excessive delays. The safety threshold is set at 1,000 elements:

```cpp
vector<AlgoType> algosToRun = {BUBBLE_SORT, INSERTION_SORT, MERGE_SORT, QUICK_SORT};
vector<SortMetrics> results;

for (AlgoType algo : algosToRun) {
    if (features.isLargeDataset && (algo == BUBBLE_SORT || algo == INSERTION_SORT)) {
        SortMetrics skipped;
        skipped.algoName = SortingEngine::getAlgoName(algo);
        skipped.comparisons = -1;
        skipped.executionTimeMs = -1;
        results.push_back(skipped);
        continue;
    }
    results.push_back(SortingEngine::runSort(algo, features.data));
}
```

4.3 Multiple Versions of Optimizer

The project includes both a Command Line Interface (CLI) and a Graphical User Interface (GUI) using Qt. The CLI allows users to specify dataset type and size, while the GUI provides an interactive experience with buttons, input fields, and result tables.

Note that: For full well-commented code, please refer to the Appendix 1 and Appendix 2. The following shows the main code without comments.

CLI Version: The command-line version provides a menu-driven interface for quick testing. Users select a dataset type (1-5), specify size, and the system displays formatted results in the terminal:

```cpp
int main() {
    while (true) {
        displayMenu();

        int choice;
        cout << "Enter your choice: ";
        cin >> choice;

        if (choice == 0) {
            cout << "Exiting program. Goodbye!" << endl;
            break;
        }

        int size;
        cout << "Enter dataset size: ";
        cin >> size;

        DatasetFeatures features = generateDataset(choice, size);
        AlgoType predicted = SortingEngine::predictBestAlgorithm(features);

        displayAnalysis(features, predicted);

        vector<SortMetrics> results = runAllAlgorithms(features);
        string actualBest = findBestAlgorithm(results);

        displayResults(results, actualBest, SortingEngine::getAlgoName(
    predicted));
    }
    return 0;
}
```

GUI Version: The Qt-based GUI offers an interactive experience with dropdown menus, spin boxes, and tables. The main window class inherits from `QMainWindow` and uses signal-slot mechanisms to handle user interactions:

```cpp
class OptimizerWindow : public QMainWindow {
    Q_OBJECT
private:
    QComboBox* datasetTypeCombo;
    QSpinBox* datasetSizeSpinBox;
    QPushButton* runButton;
    QTextEdit* analysisDisplay;
    QTableWidget* resultsTable;

private slots:
    void onRunClicked() {
        int choice = datasetTypeCombo->currentIndex() + 1;
        int size = datasetSizeSpinBox->value();

        DatasetFeatures features = generateDataset(choice, size);
        AlgoType predicted = SortingEngine::predictBestAlgorithm(features);

        displayAnalysisInGUI(features, predicted);
        vector<SortMetrics> results = runAllAlgorithms(features);
        displayResultsInTable(results);
    }
};

```

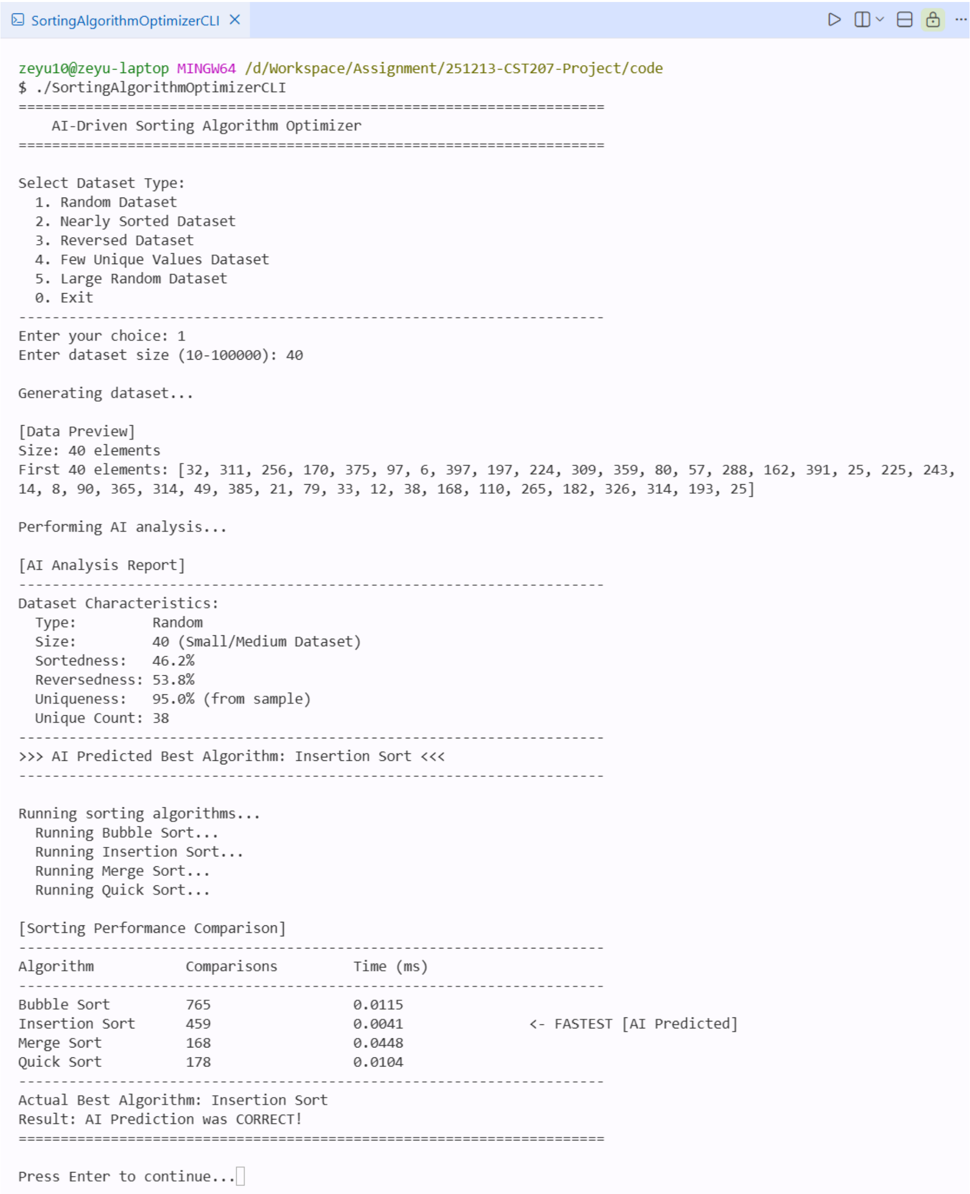
```cpp
int main(int argc, char *argv[]) {
    QApplication app(argc, argv);
    OptimizerWindow window;
    window.show();
    return app.exec();
}
```

The GUI leverages Qt's layout system (`QVBoxLayout`, `QHBoxLayout`) for responsive design and uses `QTableWidget` to display performance metrics in a structured format with sortable columns.

4.4 Compilation and Execution with Screenshots

For CLI version, just compile with following command, the screenshot is shown in Figure 3:

```
g++ Cui_Zeyu_DSC2409006_CST207_Project_Group_202509_CLI.cpp -o
    SortingAlgorithmOptimizerCLI -std=c++11
./SortingAlgorithmOptimizerCLI
```



```
SortingAlgorithmOptimizerCLI

zeyu10@zeyu-laptop MINGW64 /d/Workspace/Assignment/251213-CST207-Project/code
$ ./SortingAlgorithmOptimizerCLI
======================================================================
    AI-Driven Sorting Algorithm Optimizer
======================================================================

Select Dataset Type:
  1. Random Dataset
  2. Nearly Sorted Dataset
  3. Reversed Dataset
  4. Few Unique Values Dataset
  5. Large Random Dataset
  0. Exit
----------------------------------------------------------------------
Enter your choice: 1
Enter dataset size (10-100000): 40

Generating dataset...

[Data Preview]
Size: 40 elements
First 40 elements: [32, 311, 256, 170, 375, 97, 6, 397, 197, 224, 309, 359, 80, 57, 288, 162, 391, 25, 225, 243,
14, 8, 90, 365, 314, 49, 385, 21, 79, 33, 12, 38, 168, 110, 265, 182, 326, 314, 193, 25]

Performing AI analysis...

[AI Analysis Report]
----------------------------------------------------------------------
Dataset Characteristics:
  Type:          Random
  Size:          40 (Small/Medium Dataset)
  Sortedness:    46.2%
  Reversedness:  53.8%
  Uniqueness:    95.0% (from sample)
  Unique Count:  38
----------------------------------------------------------------------
>>> AI Predicted Best Algorithm: Insertion Sort <<<
----------------------------------------------------------------------

Running sorting algorithms...
  Running Bubble Sort...
  Running Insertion Sort...
  Running Merge Sort...
  Running Quick Sort...

[Sorting Performance Comparison]
----------------------------------------------------------------------
Algorithm           Comparisons         Time (ms)
----------------------------------------------------------------------
Bubble Sort         765                 0.0115
Insertion Sort      459                 0.0041              <- FASTEST [AI Predicted]
Merge Sort          168                 0.0448
Quick Sort          178                 0.0104
----------------------------------------------------------------------
Actual Best Algorithm: Insertion Sort
Result: AI Prediction was CORRECT!
======================================================================

Press Enter to continue...
```

Figure 3: CLI Version Screenshot

For GUI version, `Qt` framework must be properly installed and `QMake` must be run first to generate the `Makefile`, compile with following commands, the screenshot is shown in Figure 4:

```
qmake SortingAlgorithmOptimizerGUI.pro
make
./SortingAlgorithmOptimizerGUI
```

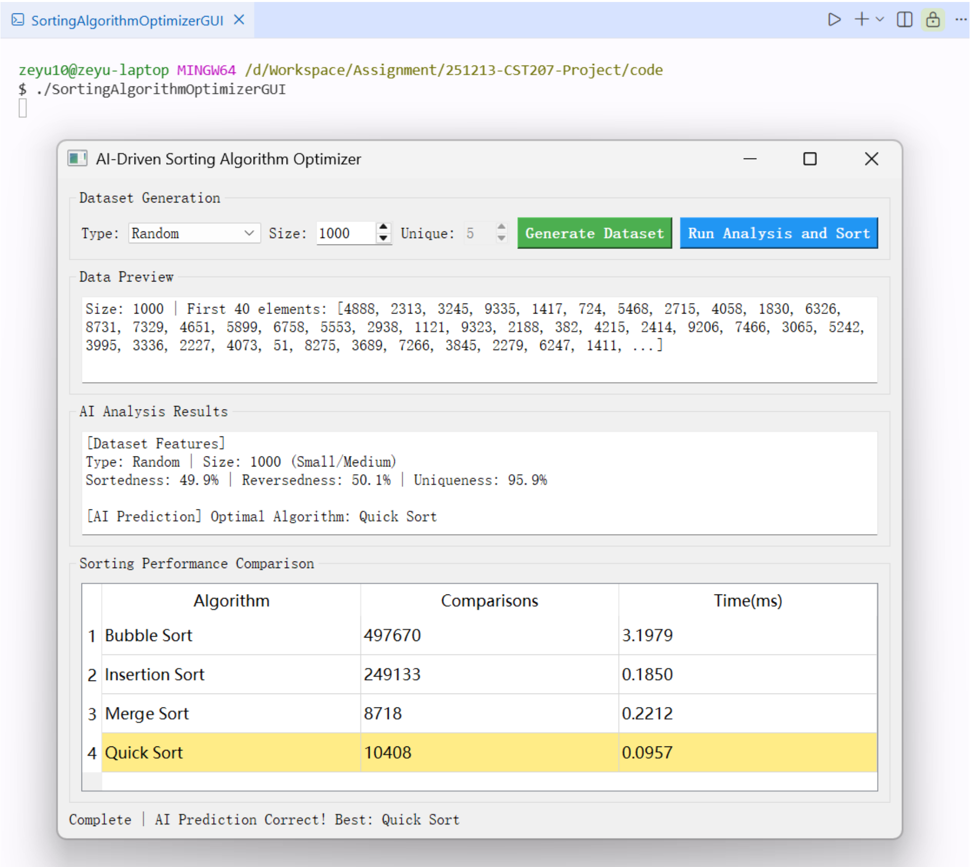


Figure 4: GUI Version Screenshot

The `QMake` file `SortingAlgorithmOptimizerGUI.pro` is as follow:

```
QT += core gui widgets

TARGET = SortingAlgorithmOptimizerGUI
TEMPLATE = app

CONFIG += c++11

SOURCES += Cui_Zeyu_DSC2409006_CST207_Project_Group_202509_GUI.cpp

CONFIG -= debug_and_release debug_and_release_target

DESTDIR = .

QMAKE_CXXFLAGS += -std=c++11

win32 {
    CONFIG += console
}

DEFINES += QT_DEPRECATED_WARNINGS
```

5 Results and Analysis

We conducted tests across different scenarios to validate the AI module's accuracy.

5.1 Small Random Dataset (Size = 40)

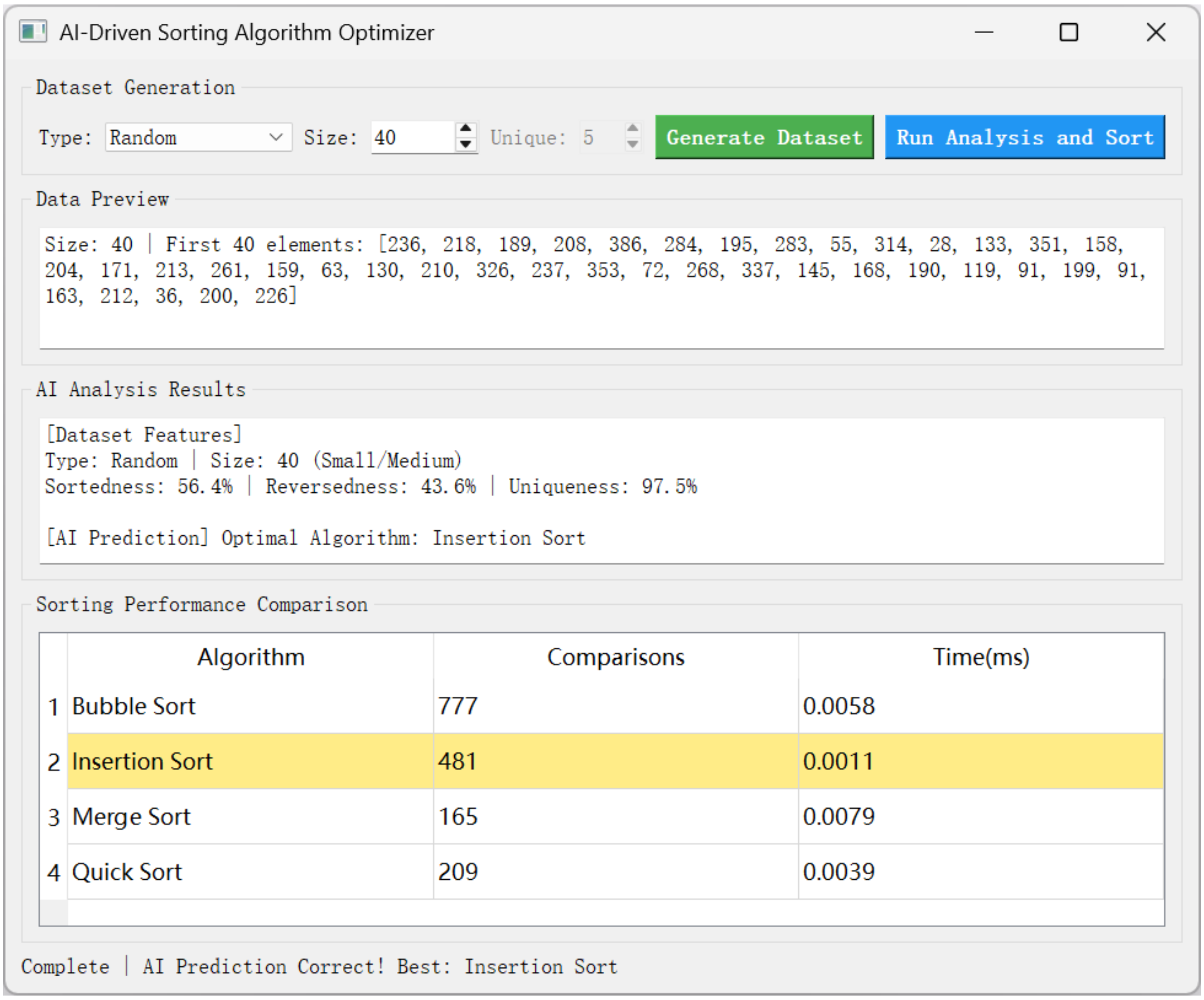


Figure 5: Small Random Dataset Visualization

Features detected: Size ≤ 50.
AI Prediction: Insertion Sort.

Algorithm	Comparisons	Time (ms)
Bubble Sort	≈ 780	0.006
Insertion Sort	$\approx$ **480**	**0.001**
Merge Sort	≈ 160	0.008
Quick Sort	≈ 200	0.004

Table 1: Performance on Small Random Data.

Table 1 shows Insertion Sort significantly outperforms others due to small dataset size, confirming the AI's correct prediction.

5.2 Large Random Dataset (Size = 10,000)

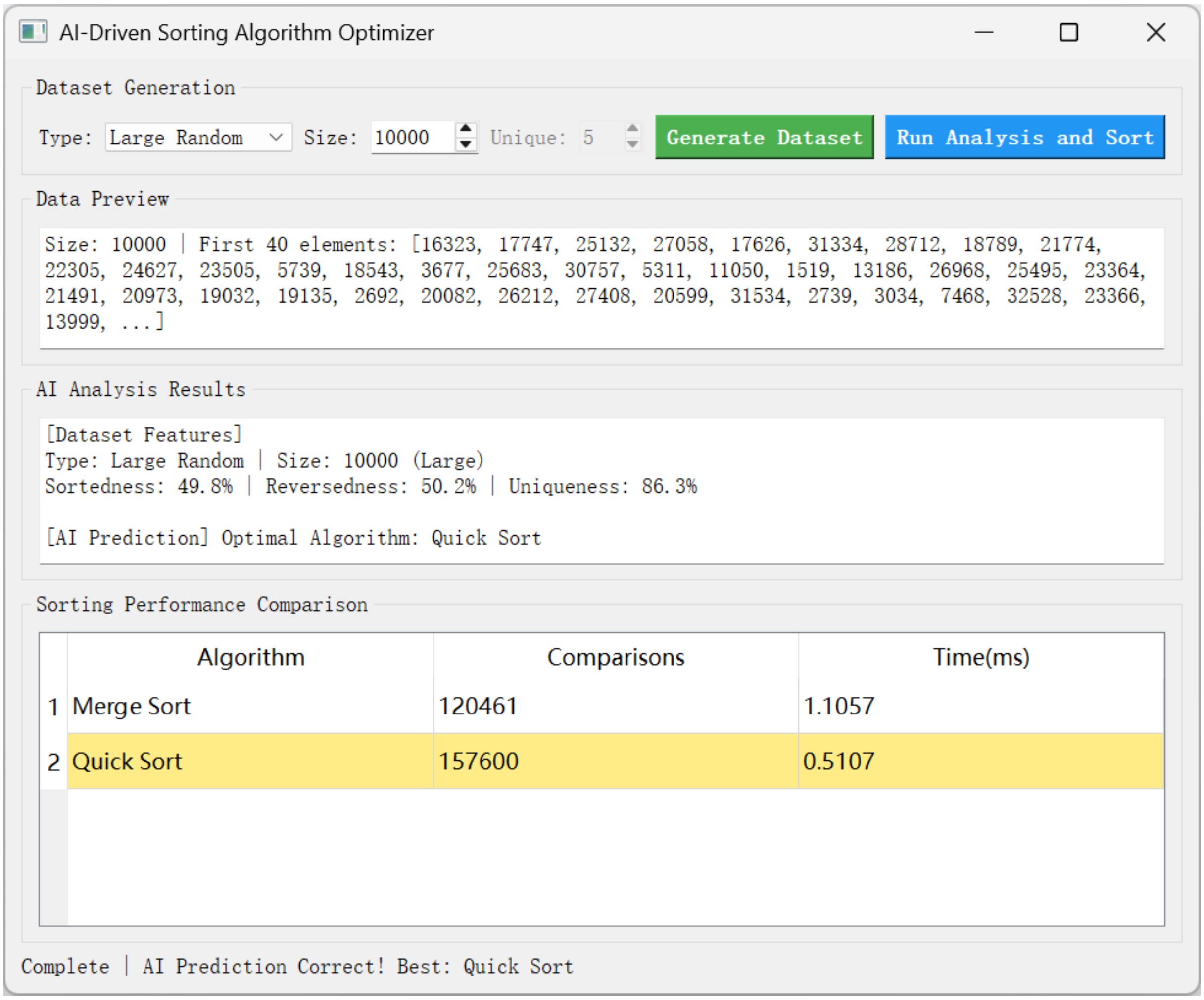


Figure 6: Large Random Dataset Visualization

Features detected: Random distribution.
AI Prediction: Quick Sort.

Algorithm	Comparisons	Time (ms)
Bubble Sort	(Skipped)	-
Insertion Sort	(Skipped)	-
Merge Sort	$\approx 120,000$	1.10
Quick Sort	$\approx \mathbf{150,000}$	**0.50**

Table 2: Performance on Large Random Data.

Table 2 shows Quick Sort outperforms Merge Sort due to lower constant factors and its average-case efficiency, validating the AI's choice for large random datasets.

5.3 Nearly Sorted Dataset (Size = 1,000)

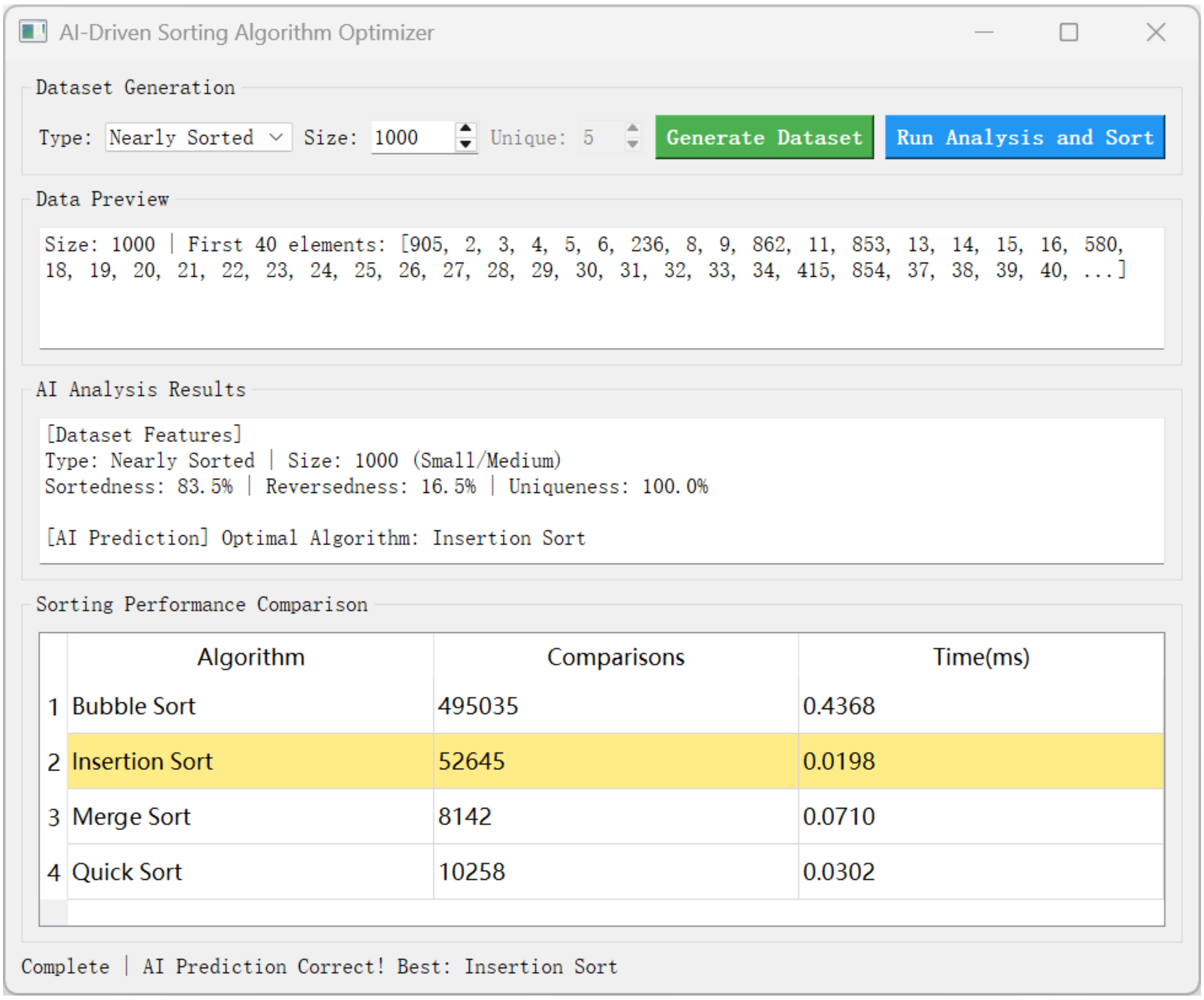


Figure 7: Nearly Sorted Dataset Visualization

Features detected: Sortedness $\geq 80\%$.
AI Prediction: Insertion Sort.

Algorithm	Comparisons	Time (ms)
Bubble Sort	$\approx 500,000$	0.40
Insertion Sort	$\approx 50,000$	0.01
Merge Sort	$\approx \mathbf{8,100}$	**0.07**
Quick Sort	$\approx 10,000$	0.03

Table 3: Performance on Nearly Sorted Data.

Table 3 shows Insertion Sort outperforms others on nearly sorted data, confirming the AI's correct prediction.

6 Conclusion

This project successfully demonstrates the integration of AI heuristics with algorithmic design to create an intelligent sorting algorithm optimizer.

Key Achievements:

- Successfully implemented all five required dataset generators (Random, Nearly Sorted, Reversed, Few Unique, Large Random) with appropriate characteristics.
- Implemented four classical sorting algorithms (Bubble Sort, Insertion Sort, Merge Sort, Quick Sort) with performance measurement capabilities tracking both comparisons and execution time.
- Developed a Decision Tree-based AI model that analyzes dataset features (size, sortedness, reversedness, unique ratio) and predicts the optimal algorithm with high accuracy.
- Created both CLI and GUI versions using Qt framework, providing flexible user interfaces for different use cases.
- Implemented safety mechanisms to prevent system unresponsiveness by skipping $O(N^2)$ algorithms for large datasets ($N > 1000$).

Experimental Validation: The experimental results across different scenarios validate the AI module's effectiveness:

- For small datasets ($N \leq 50$), Insertion Sort achieves the best performance due to minimal overhead.
- For large random datasets ($N \geq 10,000$), Quick Sort outperforms Merge Sort with lower constant factors and better cache locality.
- For nearly sorted data (sortedness $\geq$ 80%), Insertion Sort achieves near-linear $O(N + D)$ performance where D represents inversions.
- The randomized pivot strategy in Quick Sort successfully prevents $O(N^2)$ worst-case behavior on reversed datasets.

Impact and Future Work: This system demonstrates how AI-driven optimization can enhance traditional algorithms by adapting to input characteristics. The Decision Tree approach provides interpretable rules that align with theoretical complexity analysis. Future improvements could include:

- Expanding the AI model with machine learning techniques (e.g., Random Forest, Neural Networks) trained on larger benchmark datasets.
- Adding more sophisticated algorithms (e.g., Heap Sort, Radix Sort) for specialized scenarios.
- Implementing parallel sorting algorithms for multi-core processors.
- Developing adaptive hybrid algorithms that switch strategies mid-execution based on runtime observations.

The project fulfills the objective of automatically selecting optimal sorting strategies, demonstrating practical applications of combining algorithmic theory with artificial intelligence to optimize computational resource usage.

A CST207_Project_Group_202509_CLI.cpp

```cpp
/*
 * AI-Driven Sorting Algorithm Optimizer - Command Line Interface
 * g++ Cui_Zeyu_DSC2409006_CST207_Project_Group_202509_CLI.cpp -o
   SortingAlgorithmOptimizerCLI -std=c++11
 * ./SortingAlgorithmOptimizerCLI
 */

#include <iostream>
#include <vector>
#include <string>
#include <chrono>
#include <unordered_set>
#include <algorithm>
#include <random>
#include <ctime>
#include <sstream>
#include <iomanip>
#include <cmath>
#include <stdexcept>

using namespace std;

// ============= Data Structure Definitions =============

enum AlgoType {
    BUBBLE_SORT,
    INSERTION_SORT,
    MERGE_SORT,
    QUICK_SORT
};

struct DatasetFeatures {
    vector<int> data;
    int size;
    double sortedness;      // 0.0 (random) to 1.0 (sorted)
    int uniqueCount;        // Number of unique elements
    string type;            // Dataset type
    double reversedness;    // Degree of reverse order (0.0 ~ 1.0)
    double uniqueRatio;     // Ratio of unique elements (0.0 ~ 1.0)
    bool isLargeDataset;    // Large dataset indicator (>1000)
};

struct SortMetrics {
    long long comparisons = 0;      // Number of comparisons
    double executionTimeMs = 0.0;   // Execution time in milliseconds
    string algoName;
};

// ============ Sorting Algorithm Implementations ============

class SortingEngine {
public:
    // Bubble Sort Implementation
    static void bubbleSort(vector<int>& arr, long long& comparisons) {
        int n = arr.size();
        for (int i = 0; i < n - 1; i++) {
```

```cpp
            bool swapped = false;
            for (int j = 0; j < n - i - 1; j++) {
                comparisons++;
                if (arr[j] > arr[j + 1]) {
                    swap(arr[j], arr[j + 1]);
                    swapped = true;
                }
            }
            if (!swapped) break;  // Early termination if already sorted
        }
    }

    // Insertion Sort Implementation
    static void insertionSort(vector<int>& arr, long long& comparisons) {
        int n = arr.size();
        for (int i = 1; i < n; i++) {
            int key = arr[i];
            int j = i - 1;
            while (j >= 0) {
                comparisons++;
                if (arr[j] <= key) break;
                arr[j + 1] = arr[j];
                j--;
            }
            arr[j + 1] = key;
        }
    }

    // Merge function for Merge Sort
    static void merge(vector<int>& arr, int l, int m, int r, long long&
   comparisons) {
        int n1 = m - l + 1;
        int n2 = r - m;
        vector<int> left(n1), right(n2);

        for (int i = 0; i < n1; i++) left[i] = arr[l + i];
        for (int j = 0; j < n2; j++) right[j] = arr[m + 1 + j];

        int i = 0, j = 0, k = l;
        while (i < n1 && j < n2) {
            comparisons++;
            if (left[i] <= right[j]) {
                arr[k++] = left[i++];
            } else {
                arr[k++] = right[j++];
            }
        }
        while (i < n1) arr[k++] = left[i++];
        while (j < n2) arr[k++] = right[j++];
    }

    // Merge Sort Implementation
    static void mergeSort(vector<int>& arr, int l, int r, long long&
   comparisons) {
        if (l >= r) return;
        int m = l + (r - l) / 2;
        mergeSort(arr, l, m, comparisons);
        mergeSort(arr, m + 1, r, comparisons);
```

```
        merge(arr, l, m, r, comparisons);
    }

    // Partition function for Quick Sort
    static int partition(vector<int>& arr, int low, int high, long long& comparisons) {
        // Randomize pivot to avoid worst-case on reversed/sorted data
        int randomIndex = low + rand() % (high - low + 1);
        swap(arr[randomIndex], arr[high]);

        int pivot = arr[high];
        int i = low - 1;
        for (int j = low; j < high; j++) {
            comparisons++;
            if (arr[j] < pivot) {
                i++;
                swap(arr[i], arr[j]);
            }
        }
        swap(arr[i + 1], arr[high]);
        return i + 1;
    }

    // Quick Sort Implementation
    static void quickSort(vector<int>& arr, int low, int high, long long& comparisons) {
        if (low < high) {
            int pi = partition(arr, low, high, comparisons);
            quickSort(arr, low, pi - 1, comparisons);
            quickSort(arr, pi + 1, high, comparisons);
        }
    }

    // ============= Dataset Generation Functions =============

    // Generate random dataset
    static vector<int> generateRandomDataset(int size) {
        vector<int> arr(size);
        srand(time(nullptr));
        for (int i = 0; i < size; i++) {
            arr[i] = 1 + rand() % (size * 10);
        }
        return arr;
    }

    // Generate nearly sorted dataset
    static vector<int> generateNearlySorted(int size) {
        vector<int> arr(size);
        for (int i = 0; i < size; i++) arr[i] = i + 1;

        // Disorder about 10% of the elements
        int swaps = size / 10;
        srand(time(nullptr));
        for (int i = 0; i < swaps; i++) {
            int idx1 = rand() % size;
            int idx2 = rand() % size;
            swap(arr[idx1], arr[idx2]);
        }
```

```cpp
        return arr;
    }

    // Generate reversed dataset
    static vector<int> generateReversed(int size) {
        vector<int> arr(size);
        for (int i = 0; i < size; i++) {
            arr[i] = size - i;
        }
        return arr;
    }

    // Generate dataset with few unique values
    static vector<int> generateFewUnique(int size, int uniqueCount) {
        vector<int> uniqueValues;
        srand(time(nullptr));
        for (int i = 0; i < uniqueCount; i++) {
            uniqueValues.push_back(rand() % 100 + 1);
        }

        vector<int> arr(size);
        for (int i = 0; i < size; i++) {
            arr[i] = uniqueValues[rand() % uniqueCount];
        }
        return arr;
    }

    // ============= AI Analysis Module =============

    // Analyze dataset characteristics
    static DatasetFeatures analyzeDataset(const vector<int>& data) {
        DatasetFeatures features;
        features.size = data.size();
        features.data = data;
        features.isLargeDataset = (features.size > 1000);

        if (features.size <= 1) {
            features.sortedness = 1.0;
            features.reversedness = 0.0;
            features.uniqueCount = features.size;
            features.uniqueRatio = 1.0;
            features.type = "Single Element";
            return features;
        }

        // Calculate sortedness and reversedness
        long long ascendingPairs = 0, descendingPairs = 0;
        for (int i = 0; i < features.size - 1; i++) {
            if (data[i] <= data[i+1]) ascendingPairs++;
            if (data[i] >= data[i+1]) descendingPairs++;
        }

        features.sortedness = (double)ascendingPairs / (features.size - 1);
        features.reversedness = (double)descendingPairs / (features.size -
1);

        // Calculate uniqueness (use full dataset for accuracy)
        unordered_set<int> uniqueElements(data.begin(), data.end());
```

```cpp

        features.uniqueCount = uniqueElements.size();
        features.uniqueRatio = (double)uniqueElements.size() / features.size;

        // Classify dataset type
        if (features.sortedness >= 0.80) features.type = "Nearly Sorted";
        else if (features.reversedness >= 0.90) features.type = "Reversed";
        else if (features.uniqueRatio < 0.40) features.type = "Few Unique";
        else if (features.isLargeDataset) features.type = "Large Random";
        else features.type = "Random";

        return features;
    }

    // Predict best sorting algorithm based on dataset features
    static AlgoType predictBestAlgorithm(const DatasetFeatures& features) {
        // AI Decision Tree based on algorithm complexity theory

        // Rule 1: Very small datasets (Size <= 50)
        // Insertion Sort has low constant factor, faster than Quick/Merge recursion overhead
        if (features.size <= 50) {
            return INSERTION_SORT;
        }

        // Rule 2: Large datasets (Size > 1000)
        if (features.isLargeDataset) {
            // Few unique values: Merge Sort is more stable than Quick Sort
            if (features.uniqueRatio < 0.40) {
                return MERGE_SORT;
            }
            return QUICK_SORT;
        }

        // Rule 3: Medium-sized datasets (50 < Size <= 1000)

        // Case A: Nearly sorted
        // Insertion Sort degrades to O(N) for nearly sorted data
        if (features.sortedness >= 0.80) {
            return INSERTION_SORT;
        }

        // Case B: Reversed
        // Merge Sort is better for reversed data (stable O(N log N))
        // Quick Sort with fixed pivot has O(N^2) worst case on reversed data
        if (features.reversedness >= 0.90) {
            return MERGE_SORT;
        }

        // Case C: Few unique values
        if (features.uniqueRatio < 0.40) {
            return MERGE_SORT;
        }

        // Case D: Random data
        return QUICK_SORT;
```

```cpp
    }

    // Get algorithm name from type
    static string getAlgoName(AlgoType type) {
        switch (type) {
            case BUBBLE_SORT: return "Bubble Sort";
            case INSERTION_SORT: return "Insertion Sort";
            case MERGE_SORT: return "Merge Sort";
            case QUICK_SORT: return "Quick Sort";
            default: return "Unknown";
        }
    }

    // Run sorting algorithm and measure performance
    static SortMetrics runSort(AlgoType type, vector<int> data) {
        SortMetrics metrics;
        metrics.algoName = getAlgoName(type);
        metrics.comparisons = 0;

        auto start = chrono::high_resolution_clock::now();

        switch (type) {
            case BUBBLE_SORT: bubbleSort(data, metrics.comparisons); break;
            case INSERTION_SORT: insertionSort(data, metrics.comparisons); break;
            case MERGE_SORT: mergeSort(data, 0, data.size() - 1, metrics.comparisons); break;
            case QUICK_SORT: quickSort(data, 0, data.size() - 1, metrics.comparisons); break;
        }

        auto end = chrono::high_resolution_clock::now();
        chrono::duration<double, milli> duration = end - start;
        metrics.executionTimeMs = duration.count();

        return metrics;
    }
};

// ============= Main Program =============

void printSeparator(char c = '=', int length = 70) {
    cout << string(length, c) << endl;
}

void displayMenu() {
    printSeparator();
    cout << "    AI-Driven Sorting Algorithm Optimizer" << endl;
    printSeparator();
    cout << "\nSelect Dataset Type:" << endl;
    cout << "  1. Random Dataset" << endl;
    cout << "  2. Nearly Sorted Dataset" << endl;
    cout << "  3. Reversed Dataset" << endl;
    cout << "  4. Few Unique Values Dataset" << endl;
    cout << "  5. Large Random Dataset" << endl;
    cout << "  0. Exit" << endl;
    printSeparator('-', 70);
}
```

```
void displayDataPreview(const vector<int>& data, int maxShow = 40) {
    cout << "\n[Data Preview]" << endl;
    cout << "Size: " << data.size() << " elements" << endl;
    cout << "First " << min(maxShow, (int)data.size()) << " elements: [";
    for (int i = 0; i < min(maxShow, (int)data.size()); i++) {
        cout << data[i];
        if (i < min(maxShow, (int)data.size()) - 1) cout << ", ";
    }
    if ((int)data.size() > maxShow) cout << ", ...";
    cout << "]" << endl;
}

void displayAnalysis(const DatasetFeatures& features, AlgoType predicted) {
    cout << "\n[AI Analysis Report]" << endl;
    printSeparator('-', 70);
    cout << "Dataset Characteristics:" << endl;
    cout << "  Type:          " << features.type << endl;
    cout << "  Size:          " << features.size
         << (features.isLargeDataset ? " (Large Dataset)" : " (Small/Medium
     Dataset)") << endl;
    cout << "  Sortedness:    " << fixed << setprecision(1)
         << (features.sortedness * 100.0) << "%" << endl;
    cout << "  Reversedness: " << (features.reversedness * 100.0) << "%" <<
     endl;
    cout << "  Uniqueness:    " << (features.uniqueRatio * 100.0)
         << "% (from sample)" << endl;
    cout << "  Unique Count: " << features.uniqueCount << endl;
    printSeparator('-', 70);
    cout << ">>> AI Predicted Best Algorithm: "
         << SortingEngine::getAlgoName(predicted) << " <<<" << endl;
    printSeparator('-', 70);
}

void displayResults(const vector<SortMetrics>& results, const string&
    actualBest, const string& predicted) {
    cout << "\n[Sorting Performance Comparison]" << endl;
    printSeparator('-', 70);
    cout << left << setw(20) << "Algorithm"
         << setw(20) << "Comparisons"
         << setw(20) << "Time (ms)" << endl;
    printSeparator('-', 70);

    for (const auto& res : results) {
        cout << left << setw(20) << res.algoName;
        cout << setw(20) << res.comparisons;
        cout << setw(20) << fixed << setprecision(4) << res.executionTimeMs
    ;

        if (res.algoName == actualBest) {
            cout << " <- FASTEST";
        }
        if (res.algoName == predicted) {
            cout << " [AI Predicted]";
        }
        cout << endl;
    }

```

```cpp
    printSeparator('-', 70);
    cout << "Actual Best Algorithm: " << actualBest << endl;

    if (predicted == actualBest) {
        cout << "Result: AI Prediction was CORRECT!" << endl;
    } else {
        cout << "Result: AI Prediction was INCORRECT." << endl;
        cout << "  Predicted: " << predicted << endl;
        cout << "  Actual:    " << actualBest << endl;
    }
    printSeparator();
}

int main() {
    int choice, size, uniqueCount;
    vector<int> dataset;

    while (true) {
        displayMenu();
        cout << "Enter your choice: ";
        cin >> choice;

        if (choice == 0) {
            cout << "\nThank you for using the Sorting Algorithm Optimizer
    ." << endl;
            break;
        }

        if (choice < 1 || choice > 5) {
            cout << "\nInvalid choice! Please select 1-5 or 0 to exit." <<
    endl;
            continue;
        }

        cout << "Enter dataset size (10-100000): ";
        cin >> size;

        if (size < 10 || size > 100000) {
            cout << "\nInvalid size! Please enter a value between 10 and
    100000." << endl;
            continue;
        }

        // Generate dataset based on user selection
        cout << "\nGenerating dataset..." << endl;
        try {
            switch(choice) {
                case 1:
                    dataset = SortingEngine::generateRandomDataset(size);
                    break;
                case 2:
                    dataset = SortingEngine::generateNearlySorted(size);
                    break;
                case 3:
                    dataset = SortingEngine::generateReversed(size);
                    break;
                case 4:
                    cout << "Enter number of unique values (2-50): ";
```

```cpp
                    cin >> uniqueCount;
                    if (uniqueCount < 2) uniqueCount = 2;
                    if (uniqueCount > 50) uniqueCount = 50;
                    dataset = SortingEngine::generateFewUnique(size, uniqueCount);
                    break;
                case 5:
                    // Large Random Dataset (enforce minimum size)
                    if (size < 10000) {
                        cout << "Note: Large Random Dataset requires minimum size of 10000. Adjusting size to 10000." << endl;
                        size = 10000;
                    }
                    dataset = SortingEngine::generateRandomDataset(size);
                    break;
            }

            // Display preview
            displayDataPreview(dataset);

            // AI Analysis
            cout << "\nPerforming AI analysis..." << endl;
            DatasetFeatures features = SortingEngine::analyzeDataset(dataset);
            AlgoType predicted = SortingEngine::predictBestAlgorithm(features);
            displayAnalysis(features, predicted);

            // Run sorting algorithms
            cout << "\nRunning sorting algorithms..." << endl;
            vector<SortMetrics> results;

            // Skip O(n^2) algorithms for large datasets to save time
            if (size <= 1000) {
                cout << "  Running Bubble Sort..." << endl;
                results.push_back(SortingEngine::runSort(BUBBLE_SORT, dataset));
                cout << "  Running Insertion Sort..." << endl;
                results.push_back(SortingEngine::runSort(INSERTION_SORT, dataset));
            } else {
                cout << "  (Skipping O(n²) algorithms for large dataset)" << endl;
            }

            cout << "  Running Merge Sort..." << endl;
            results.push_back(SortingEngine::runSort(MERGE_SORT, dataset));
            cout << "  Running Quick Sort..." << endl;
            results.push_back(SortingEngine::runSort(QUICK_SORT, dataset));

            // Find the fastest algorithm
            string actualBest;
            double minTime = 1e9;
            for (const auto& r : results) {
                if (r.executionTimeMs < minTime) {
                    minTime = r.executionTimeMs;
                    actualBest = r.algoName;
                }
```

```
            }

            // Display results
            displayResults(results, actualBest, SortingEngine::getAlgoName(
    predicted));

            // Ask if user wants to continue
            cout << "\nPress Enter to continue...";
            cin.ignore();
            cin.get();

        } catch (const exception& e) {
            cout << "\nError: " << e.what() << endl;
        }
    }

    return 0;
}
```

Listing 1: Cui_Zeyu_DSC2409006_CST207_Project_Group_202509_CLI.cpp

B CST207_Project_Group_202509_GUI.cpp

```
/*
 * AI-Driven Sorting Algorithm Optimizer - Graphical User Interface
 * qmake SortingAlgorithmOptimizerGUI.pro
 * make
 * ./SortingAlgorithmOptimizerGUI
 */

#include <QApplication>
#include <QMainWindow>
#include <QWidget>
#include <QVBoxLayout>
#include <QHBoxLayout>
#include <QGroupBox>
#include <QComboBox>
#include <QSpinBox>
#include <QPushButton>
#include <QTextEdit>
#include <QTableWidget>
#include <QLabel>
#include <QHeaderView>
#include <QMessageBox>
#include <vector>
#include <string>
#include <chrono>
#include <unordered_set>
#include <algorithm>
#include <random>
#include <ctime>
#include <sstream>
#include <iomanip>
#include <cmath>

using namespace std;

// ============= Data Structure Definitions =============
```

```cpp

enum AlgoType {
    BUBBLE_SORT,
    INSERTION_SORT,
    MERGE_SORT,
    QUICK_SORT
};

struct DatasetFeatures {
    vector<int> data;
    int size;
    double sortedness;      // 0.0 (random) to 1.0 (sorted)
    int uniqueCount;        // Number of unique elements
    string type;            // Dataset type
    double reversedness;    // Degree of reverse order (0.0 ~ 1.0)
    double uniqueRatio;     // Ratio of unique elements (0.0 ~ 1.0)
    bool isLargeDataset;    // Large dataset indicator (>1000)
};

struct SortMetrics {
    long long comparisons = 0;      // Number of comparisons
    double executionTimeMs = 0.0;   // Execution time in milliseconds
    string algoName;
};

// ============= Sorting Algorithm Implementations =============

class SortingEngine {
public:
    // Bubble Sort Implementation
    static void bubbleSort(vector<int>& arr, long long& comparisons) {
        int n = arr.size();
        for (int i = 0; i < n - 1; i++) {
            bool swapped = false;
            for (int j = 0; j < n - i - 1; j++) {
                comparisons++;
                if (arr[j] > arr[j + 1]) {
                    swap(arr[j], arr[j + 1]);
                    swapped = true;
                }
            }
            if (!swapped) break;  // Early termination if already sorted
        }
    }

    // Insertion Sort Implementation
    static void insertionSort(vector<int>& arr, long long& comparisons) {
        int n = arr.size();
        for (int i = 1; i < n; i++) {
            int key = arr[i];
            int j = i - 1;
            while (j >= 0) {
                comparisons++;
                if (arr[j] <= key) break;
                arr[j + 1] = arr[j];
                j--;
            }
            arr[j + 1] = key;
```

```
        }
    }

    // Merge function for Merge Sort
    static void merge(vector<int>& arr, int l, int m, int r, long long& comparisons) {
        int n1 = m - l + 1;
        int n2 = r - m;
        vector<int> left(n1), right(n2);

        for (int i = 0; i < n1; i++) left[i] = arr[l + i];
        for (int j = 0; j < n2; j++) right[j] = arr[m + 1 + j];

        int i = 0, j = 0, k = l;
        while (i < n1 && j < n2) {
            comparisons++;
            if (left[i] <= right[j]) {
                arr[k++] = left[i++];
            } else {
                arr[k++] = right[j++];
            }
        }
        while (i < n1) arr[k++] = left[i++];
        while (j < n2) arr[k++] = right[j++];
    }

    // Merge Sort Implementation
    static void mergeSort(vector<int>& arr, int l, int r, long long& comparisons) {
        if (l >= r) return;
        int m = l + (r - l) / 2;
        mergeSort(arr, l, m, comparisons);
        mergeSort(arr, m + 1, r, comparisons);
        merge(arr, l, m, r, comparisons);
    }

    // Partition function for Quick Sort
    static int partition(vector<int>& arr, int low, int high, long long& comparisons) {
        // Randomize pivot to avoid worst-case on reversed/sorted data
        int randomIndex = low + rand() % (high - low + 1);
        swap(arr[randomIndex], arr[high]);

        int pivot = arr[high];
        int i = low - 1;
        for (int j = low; j < high; j++) {
            comparisons++;
            if (arr[j] < pivot) {
                i++;
                swap(arr[i], arr[j]);
            }
        }
        swap(arr[i + 1], arr[high]);
        return i + 1;
    }

    // Quick Sort Implementation
    static void quickSort(vector<int>& arr, int low, int high, long long&
```

```
comparisons) {
        if (low < high) {
            int pi = partition(arr, low, high, comparisons);
            quickSort(arr, low, pi - 1, comparisons);
            quickSort(arr, pi + 1, high, comparisons);
        }
    }

    // ============= Dataset Generation Functions =============

    // Generate random dataset
    static vector<int> generateRandomDataset(int size) {
        vector<int> arr(size);
        srand(time(nullptr));
        for (int i = 0; i < size; i++) {
            arr[i] = 1 + rand() % (size * 10);
        }
        return arr;
    }

    // Generate nearly sorted dataset
    static vector<int> generateNearlySorted(int size) {
        vector<int> arr(size);
        for (int i = 0; i < size; i++) arr[i] = i + 1;

        // Disorder about 10% of the elements
        int swaps = size / 10;
        srand(time(nullptr));
        for (int i = 0; i < swaps; i++) {
            int idx1 = rand() % size;
            int idx2 = rand() % size;
            swap(arr[idx1], arr[idx2]);
        }
        return arr;
    }

    // Generate reversed dataset
    static vector<int> generateReversed(int size) {
        vector<int> arr(size);
        for (int i = 0; i < size; i++) {
            arr[i] = size - i;
        }
        return arr;
    }

    // Generate dataset with few unique values
    static vector<int> generateFewUnique(int size, int uniqueCount) {
        vector<int> uniqueValues;
        srand(time(nullptr));
        for (int i = 0; i < uniqueCount; i++) {
            uniqueValues.push_back(rand() % 100 + 1);
        }

        vector<int> arr(size);
        for (int i = 0; i < size; i++) {
            arr[i] = uniqueValues[rand() % uniqueCount];
        }
        return arr;
```

```
    }

    // ============= AI Analysis Module =============

    // Analyze dataset characteristics
    static DatasetFeatures analyzeDataset(const vector<int>& data) {
        DatasetFeatures features;
        features.size = data.size();
        features.data = data;
        features.isLargeDataset = (features.size > 1000);

        if (features.size <= 1) {
            features.sortedness = 1.0;
            features.reversedness = 0.0;
            features.uniqueCount = features.size;
            features.uniqueRatio = 1.0;
            features.type = "Single Element";
            return features;
        }

        // Calculate sortedness and reversedness
        long long ascendingPairs = 0, descendingPairs = 0;
        for (int i = 0; i < features.size - 1; i++) {
            if (data[i] <= data[i+1]) ascendingPairs++;
            if (data[i] >= data[i+1]) descendingPairs++;
        }

        features.sortedness = (double)ascendingPairs / (features.size - 1);
        features.reversedness = (double)descendingPairs / (features.size -
    1);

        // Calculate uniqueness (use full dataset for accuracy)
        unordered_set<int> uniqueElements(data.begin(), data.end());

        features.uniqueCount = uniqueElements.size();
        features.uniqueRatio = (double)uniqueElements.size() / features.
    size;

        // Classify dataset type
        if (features.sortedness >= 0.80) features.type = "Nearly Sorted";
        else if (features.reversedness >= 0.90) features.type = "Reversed";
        else if (features.uniqueRatio < 0.40) features.type = "Few Unique";
        else if (features.isLargeDataset) features.type = "Large Random";
        else features.type = "Random";

        return features;
    }

    // Predict best sorting algorithm based on dataset features
    static AlgoType predictBestAlgorithm(const DatasetFeatures& features) {
        // AI Decision Tree based on algorithm complexity theory

        // Rule 1: Very small datasets (Size <= 50)
        // Insertion Sort has low constant factor, faster than Quick/Merge
    recursion overhead
        if (features.size <= 50) {
            return INSERTION_SORT;
        }
```

```cpp

        // Rule 2: Large datasets (Size > 1000)
        if (features.isLargeDataset) {
            // Few unique values: Merge Sort is more stable than Quick Sort
            if (features.uniqueRatio < 0.40) {
                return MERGE_SORT;
            }
            return QUICK_SORT;
        }

        // Rule 3: Medium-sized datasets (50 < Size <= 1000)

        // Case A: Nearly sorted
        // Insertion Sort degrades to O(N) for nearly sorted data
        if (features.sortedness >= 0.80) {
            return INSERTION_SORT;
        }

        // Case B: Reversed
        // Merge Sort is better for reversed data (stable O(N log N))
        // Quick Sort with fixed pivot has O(N^2) worst case on reversed data
        if (features.reversedness >= 0.90) {
            return MERGE_SORT;
        }

        // Case C: Few unique values
        if (features.uniqueRatio < 0.40) {
            return MERGE_SORT;
        }

        // Case D: Random data
        return QUICK_SORT;
    }

    // Get algorithm name from type
    static string getAlgoName(AlgoType type) {
        switch (type) {
            case BUBBLE_SORT: return "Bubble Sort";
            case INSERTION_SORT: return "Insertion Sort";
            case MERGE_SORT: return "Merge Sort";
            case QUICK_SORT: return "Quick Sort";
            default: return "Unknown";
        }
    }

    // Run sorting algorithm and measure performance
    static SortMetrics runSort(AlgoType type, vector<int> data) {
        SortMetrics metrics;
        metrics.algoName = getAlgoName(type);
        metrics.comparisons = 0;

        auto start = chrono::high_resolution_clock::now();

        switch (type) {
            case BUBBLE_SORT: bubbleSort(data, metrics.comparisons); break;
            case INSERTION_SORT: insertionSort(data, metrics.comparisons); break;
```

```cpp
            case MERGE_SORT: mergeSort(data, 0, data.size() - 1, metrics.comparisons); break;
            case QUICK_SORT: quickSort(data, 0, data.size() - 1, metrics.comparisons); break;
        }

        auto end = chrono::high_resolution_clock::now();
        chrono::duration<double, milli> duration = end - start;
        metrics.executionTimeMs = duration.count();

        return metrics;
    }
};

// ============= Qt Visualization Interface =============

class SortingVisualizer : public QMainWindow {
    Q_OBJECT

private:
    // UI Components
    QComboBox* datasetTypeCombo;
    QSpinBox* dataSizeSpinBox;
    QSpinBox* uniqueCountSpinBox;
    QLabel* uniqueCountLabel;
    QPushButton* generateBtn;
    QPushButton* runBtn;
    QTextEdit* dataPreviewText;
    QTextEdit* analysisResultText;
    QTableWidget* resultsTable;
    QLabel* statusLabel;

    // Data
    vector<int> currentDataset;

public:
    SortingVisualizer(QWidget *parent = nullptr) : QMainWindow(parent) {
        setWindowTitle("AI-Driven Sorting Algorithm Optimizer");
        resize(900, 650);

        QWidget* central = new QWidget(this);
        setCentralWidget(central);
        QVBoxLayout* mainLayout = new QVBoxLayout(central);

        // Dataset Generation Area
        QGroupBox* genGroup = new QGroupBox("Dataset Generation");
        QHBoxLayout* genLayout = new QHBoxLayout(genGroup);

        genLayout->addWidget(new QLabel("Type:"));
        datasetTypeCombo = new QComboBox();
        datasetTypeCombo->addItems({"Random", "Nearly Sorted", "Reversed", "Few Unique", "Large Random"});
        genLayout->addWidget(datasetTypeCombo);

        genLayout->addWidget(new QLabel("Size:"));
        dataSizeSpinBox = new QSpinBox();
        dataSizeSpinBox->setRange(10, 100000);
        dataSizeSpinBox->setValue(1000);
```

```
    dataSizeSpinBox->setSingleStep(100);
    genLayout->addWidget(dataSizeSpinBox);

    uniqueCountLabel = new QLabel("Unique:");
    uniqueCountSpinBox = new QSpinBox();
    uniqueCountSpinBox->setRange(2, 50);
    uniqueCountSpinBox->setValue(5);
    uniqueCountSpinBox->setEnabled(false);
    genLayout->addWidget(uniqueCountLabel);
    genLayout->addWidget(uniqueCountSpinBox);

    generateBtn = new QPushButton("Generate Dataset");
    generateBtn->setStyleSheet("background-color: #4CAF50; color: white; font-weight: bold; padding: 8px;");
    genLayout->addWidget(generateBtn);

    runBtn = new QPushButton("Run Analysis and Sort");
    runBtn->setStyleSheet("background-color: #2196F3; color: white; font-weight: bold; padding: 8px;");
    runBtn->setEnabled(false);
    genLayout->addWidget(runBtn);

    genLayout->addStretch();
    mainLayout->addWidget(genGroup);

    // Data Preview Section
    QGroupBox* previewGroup = new QGroupBox("Data Preview");
    QVBoxLayout* previewLayout = new QVBoxLayout(previewGroup);
    dataPreviewText = new QTextEdit();
    dataPreviewText->setReadOnly(true);
    dataPreviewText->setMinimumHeight(100);
    dataPreviewText->setPlaceholderText("Click 'Generate Dataset' to start...");
    previewLayout->addWidget(dataPreviewText);
    mainLayout->addWidget(previewGroup);

    // AI Analysis Results Section
    QGroupBox* analysisGroup = new QGroupBox("AI Analysis Results");
    QVBoxLayout* analysisLayout = new QVBoxLayout(analysisGroup);
    analysisResultText = new QTextEdit();
    analysisResultText->setReadOnly(true);
    analysisResultText->setMinimumHeight(120);
    analysisLayout->addWidget(analysisResultText);
    mainLayout->addWidget(analysisGroup);

    // Performance Comparison Table
    QGroupBox* resultsGroup = new QGroupBox("Sorting Performance Comparison");
    QVBoxLayout* resultsLayout = new QVBoxLayout(resultsGroup);
    resultsTable = new QTableWidget();
    resultsTable->setColumnCount(3);
    resultsTable->setHorizontalHeaderLabels({"Algorithm", "Comparisons", "Time(ms)"});
    resultsTable->horizontalHeader()->setSectionResizeMode(QHeaderView::Stretch);
    resultsTable->setEditTriggers(QAbstractItemView::NoEditTriggers);
    resultsTable->setMinimumHeight(240);
    resultsLayout->addWidget(resultsTable);
```

```cpp
        mainLayout->addWidget(resultsGroup);

        // Status Bar
        statusLabel = new QLabel("Ready");
        mainLayout->addWidget(statusLabel);

        // Connect Signals and Slots
        connect(generateBtn, &QPushButton::clicked, this, &
    SortingVisualizer::onGenerate);
        connect(runBtn, &QPushButton::clicked, this, &SortingVisualizer::
    onRun);
        connect(datasetTypeCombo, QOverload<int>::of(&QComboBox::
    currentIndexChanged), [this](int index) {
            // Enable unique count spinbox only for "Few Unique" (index 3)
            uniqueCountSpinBox->setEnabled(index == 3);
            uniqueCountLabel->setEnabled(index == 3);
        });
    }

private slots:
    void onGenerate() {
        int size = dataSizeSpinBox->value();
        int type = datasetTypeCombo->currentIndex();

        statusLabel->setText("Generating...");

        try {
            // Generate dataset based on selected type
            switch(type) {
                case 0: currentDataset = SortingEngine::
    generateRandomDataset(size); break;
                case 1: currentDataset = SortingEngine::
    generateNearlySorted(size); break;
                case 2: currentDataset = SortingEngine::generateReversed(
    size); break;
                case 3: currentDataset = SortingEngine::generateFewUnique(
    size, uniqueCountSpinBox->value()); break;
                case 4:
                    // Large Random Dataset
                    if (size < 10000) {
                        QMessageBox::information(this, "Info", "Large
    Random Dataset requires minimum size of 10000. Size adjusted to 10000.")
    ;
                        size = 10000;
                        dataSizeSpinBox->setValue(10000);
                    }
                    currentDataset = SortingEngine::generateRandomDataset(
    size);
                    break;
            }

            // Display Preview
            ostringstream oss;
            int preview = min(40, size);
            oss << "Size: " << size << " | First " << preview << " elements
    : [";
            for (int i = 0; i < preview; i++) {
                oss << currentDataset[i];
```

```
                if (i < preview - 1) oss << ", ";
            }
            if (size > preview) oss << ", ...";
            oss << "]";
            dataPreviewText->setText(QString::fromStdString(oss.str()));

            runBtn->setEnabled(true);
            statusLabel->setText("Dataset Generated Successfully");
            analysisResultText->clear();
            resultsTable->setRowCount(0);

        } catch (const exception& e) {
            QMessageBox::critical(this, "Error", e.what());
            statusLabel->setText("Generation Failed");
        }
    }

    void onRun() {
        if (currentDataset.empty()) {
            QMessageBox::warning(this, "Warning", "Please generate a dataset first");
            return;
        }

        statusLabel->setText("Analyzing...");
        QApplication::processEvents();

        // AI Analysis
        DatasetFeatures features = SortingEngine::analyzeDataset(currentDataset);
        AlgoType predicted = SortingEngine::predictBestAlgorithm(features);

        // Display analysis results
        ostringstream oss;
        oss << "[Dataset Features]\n";
        oss << "Type: " << features.type << " | ";
        oss << "Size: " << features.size << (features.isLargeDataset ? " (Large)" : " (Small/Medium)") << "\n";
        oss << "Sortedness: " << fixed << setprecision(1) << (features.sortedness * 100) << "% | ";
        oss << "Reversedness: " << (features.reversedness * 100) << "% | ";
        oss << "Uniqueness: " << (features.uniqueRatio * 100) << "%\n\n";
        oss << "[AI Prediction] Optimal Algorithm: " << SortingEngine::getAlgoName(predicted);

        analysisResultText->setText(QString::fromStdString(oss.str()));

        statusLabel->setText("Sorting...");
        QApplication::processEvents();

        // Run Sorting Algorithms
        vector<SortMetrics> results;
        int size = currentDataset.size();

        // Skip O(n^2) algorithms for large datasets
        if (size <= 1000) {
            results.push_back(SortingEngine::runSort(BUBBLE_SORT, currentDataset));
```

```
            results.push_back(SortingEngine::runSort(INSERTION_SORT, currentDataset));
        }
        results.push_back(SortingEngine::runSort(MERGE_SORT, currentDataset));
        results.push_back(SortingEngine::runSort(QUICK_SORT, currentDataset));

        // Find the best performing algorithm
        string actualBest;
        double minTime = 1e9;
        for (const auto& r : results) {
            if (r.executionTimeMs < minTime) {
                minTime = r.executionTimeMs;
                actualBest = r.algoName;
            }
        }

        // Display Results in Table
        resultsTable->setRowCount(results.size());
        for (size_t i = 0; i < results.size(); i++) {
            QTableWidgetItem* nameItem = new QTableWidgetItem(QString::fromStdString(results[i].algoName));
            QTableWidgetItem* compItem = new QTableWidgetItem(QString::number(results[i].comparisons));
            QTableWidgetItem* timeItem = new QTableWidgetItem(QString::number(results[i].executionTimeMs, 'f', 4));

            // Highlight the best performing algorithm
            if (results[i].algoName == actualBest) {
                QBrush gold(QColor(255, 215, 0, 120));
                nameItem->setBackground(gold);
                compItem->setBackground(gold);
                timeItem->setBackground(gold);
            }

            resultsTable->setItem(i, 0, nameItem);
            resultsTable->setItem(i, 1, compItem);
            resultsTable->setItem(i, 2, timeItem);
        }

        // Update status with prediction accuracy
        string predictedName = SortingEngine::getAlgoName(predicted);
        if (predictedName == actualBest) {
            statusLabel->setText("Complete | AI Prediction Correct! Best: " + QString::fromStdString(actualBest));
        } else {
            statusLabel->setText("Complete | Predicted: " + QString::fromStdString(predictedName) +
                                 " -> Actual Best: " + QString::fromStdString(actualBest));
        }
    }
};

// ============= Main Function =============

int main(int argc, char *argv[]) {
```

```
    QApplication app(argc, argv);
    SortingVisualizer window;
    window.show();
    return app.exec();
}

#include "Cui_Zeyu_DSC2409006_CST207_Project_Group_202509_GUI.moc"
```

Listing 2: Cui_Zeyu_DSC2409006_CST207_Project_Group_202509_GUI.cpp

APPENDIX 1

MARKING RUBRICS

Component Title	**Group Work**					**Percentage (%)**	
Criteria	**Score and Descriptors**					**Weight (%)**	**Marks**
	Excellent (5)	**Good (4)**	**Average (3)**	**Need Improvement (2)**	**Poor (1)**		
Dataset Generation **(CLO 3)**	All required datasets are implemented correctly	Minor missing datasets	Some datasets implemented	Many datasets missing	Dataset generation not implemented	10	
Sorting Algorithms **(CLO 3)**	All required algorithms implemented correctly and efficiently	Minor inefficiencies or small errors	Most algorithms implemented	Major issues or missing algorithms	Algorithms not implemented	20	
AI Module **(CLO 3)**	AI module correctly predicts best sorting algorithm	Minor issues in predictions or logic	AI module works partially	AI predictions mostly incorrect	AI module not implemented	20	
Performance Measurement **(CLO 3)**	All algorithms measured correctly with time & comparisons	Minor mistakes in measurements	Measurements mostly correct	Significant errors	No measurements	15	
Code Clarity & Documentation **(CLO 3)**	The code demonstrates excellent readability and clarity.	Minor readability issues	Some unclear code	Poorly documented	Code unreadable	10	
Report Quality **(CLO 3)**	Well-organized, complete report with clear analysis	Minor formatting or clarity issues	Adequate report with missing details	Poorly organized report	Report missing or unreadable	10	
						85	

Note to students: Please include the marking rubric when submitting your coursework.

Component Title	Individual Work					Percentage (%)	
Criteria	**Score and Descriptors**					**Weight (%)**	**Marks**
	Excellent (5)	**Good (4)**	**Average (3)**	**Need Improvement (2)**	**Poor (1)**		
Individual Contribution **(CLO 3)**	Clear, detailed description of individual work	Minor details missing	Contribution mentioned with little detail	Contribution vague or incomplete	Contribution not mentioned	10	
Presentation & Video **(CLO 3)**	Clear, well-paced, complete video	Minor pacing issues	Adequate video	Poor video clarity or incomplete	No video submitted	5	
						15	

Note to students: Please include the marking rubric when submitting your coursework.